UNITED STATES PATENT APPLICATION

FOR

PARALLEL APPLY PROCESSING IN DATA REPLICATION WITH PRESERVATION OF TRANSACTION INTEGRITY AND SOURCE ORDERING OF DEPENDENT UPDATES

Inventor(s):
Serge BOURBONNAIS
Elizabeth B. HAMEL
Bruce G. LINDSAY
Stephen J. TODD

Sawyer Law Group LLP
2465 E. Bayshore Road, Suite 406
Palo Alto, California 94303

PARALLEL APPLY PROCESSING IN DATA REPLICATION WITH PRESERVATION OF TRANSACTION INTEGRITY AND SOURCE ORDERING OF DEPENDENT UPDATES

FIELD OF THE INVENTION

The present invention relates to the maintenance of multiple copies of tabular data, and more particularly to providing parallelized apply of asynchronously replicated transactional changes to a target database.

BACKGROUND OF THE INVENTION

In a relational database management system, data is stored in a multiplicity of tables having a multiplicity of rows (records), the rows having a multiplicity of columns (fields). A subset of the columns are designated as key columns and the combination of values of the key columns of the rows of a single table must be distinct. It is frequently desired to maintain copies (replicas) of a first table residing in a first database of the relational variety in one or more other databases of the relational variety. Furthermore, it is desired that changes (inserts, deletes, and updates) to rows of the table in the first database be copied (replicated) to the table copies residing in the other databases. Additionally, it is sometimes desired that the changes made to any of the table copies residing in any of the several relational databases be copied (replicated) to all the other table copies.

The propagation of changes made to one copy of the table may be synchronous or asynchronous to the original change. Synchronous propagation makes changes at all copies as part of the same transaction (unit of work) that initiates the original changes.

Asynchronous propagation copies the original changes to the other table copies in separate transactions, subsequent to the completion of the transaction initiating the original changes. Synchronous change propagation requires that the database management systems maintaining all (or most) copies be active and available at the time of the change. Also, synchronous change propagation introduces substantial messaging and synchronization costs at the time of the original changes.

The means of detecting changes to be propagated asynchronously can be active or passive. Active change detection isolates the changes, at the time of the change, for later processing using database triggers or a similar mechanism. Passive change detection exploits information from the database recovery log, where changes are recorded for other purposes, to deduce what rows, of which tables, were changed as well as both the old and new values of changed columns.

In a typical database environment, there are varying levels of parallel transactional processing, involving concurrent transactions that execute read and write actions against database information. Fundamental to the nature of a data replication process is the choice of how to move, order and apply that stream of parallel database event changes to a target database.

One conventional approach provides a certain degree of apply parallelism by grouping related tables into distinct sets and having each set of tables applied by a completely separate program. However, this approach places a heavy burden the user, who may have difficulty knowing which tables are logically related and must be grouped

together.

In another conventional approach, parallelism is provided but without preserving the source data event order. Thus, to provide data integrity, a "shadow" table is used to track and maintain each individual data row change. This approach, however, has a significant overhead cost in both making updates and in performing lookups against the shadow table.

Other conventional approaches provide parallelism but by using a very proprietary way that has no or limited applicability outside of a specific system.

Accordingly, there exists a need for an improved method for providing parallel apply in asynchronous data replication in a database system. The improved method and system should provide a high speed parallel apply of transactional changes to a target node such that the parallel nature of the application of changes does not compromise the integrity of the data. The improved method and system should also require significantly less overhead than conventional approaches and be easily adaptable to various types of database systems. The present invention addresses such a need.

SUMMARY OF THE INVENTION

An improved method for providing parallel apply in asynchronous data replication in a database system is disclosed. The improved method and system provides a high speed parallel apply of transactional changes to a target node such that the parallel nature of the application of changes does not compromise the integrity of the data. The method and system detects, tracks, and handles dependencies between transaction messages to be applied

to the target node. If a transaction message has a dependency on one or more preceding transaction messages whose applications have not yet completed, that transaction message is held until the application completes. In addition, the method and system requires significantly less overhead than conventional approaches.

BRIEF DESCRIPTION OF THE FIGURES

Figure 1 illustrates an embodiment of a system for providing parallel apply in asynchronous data replication in a database system in accordance with the present invention.

Figure 2 is a flowchart illustrating an embodiment of a method for providing parallel apply in asynchronous data replication in a database system in accordance with the present invention.

Figure 3 is a flowchart illustrating in more detail the determining of dependencies in the method for providing parallel apply in asynchronous data replication in a database system in accordance with the present invention.

Figure 4 illustrates an example of the method for providing parallel apply in asynchronous data replication in a database system in accordance with the present invention.

DETAILED DESCRIPTION

The present invention provides an improved method for providing parallel apply in asynchronous data replication in a database system. The following description is presented to enable one of ordinary skill in the art to make and use the invention and is provided in the

context of a patent application and its requirements. Various modifications to the preferred embodiment will be readily apparent to those skilled in the art and the generic principles herein may be applied to other embodiments. Thus, the present invention is not intended to be limited to the embodiment shown but is to be accorded the widest scope consistent with the principles and features described herein.

To more particularly describe the features of the present invention, please refer to Figures 1 through 4 in conjunction with the discussion below.

Figure 1 illustrates an embodiment of a system for providing parallel apply in asynchronous data replication in a database system in accordance with the present invention. The system includes a source node 101 and a target node 106. At the source node 101 are one or more source table copies 102, a recovery log 103, a Capture program 104 ("Capture"), and a send queue 105. At the target node 106 are a receive queue 107, an Apply program ("Apply") 108 and one or more target table copies 112. Apply 108 includes a browser thread 109, a work queue 110, a done queue 111, and one or more agent threads 112. Capture 104 reads changes of committed transactions from the recovery log 103 and sends them to Apply 108 running on the target node 106. Apply 108 eventually re-executes the changes of the transactions.

In this embodiment of the present invention, the tabular data at the source table copies 102 whose changes are to be replicated to the target table copies 113 reside in a Relational Database management System (RDBMS) such as the DB2™ RDBMS product offered by International Business Machines Corporation™. The RDBMS maintains a

recovery log 103 and a means to query its contents. The entries of the recovery log 103 describe changes to rows of the source tables 102 at source node 101. More specifically, the entries in the recovery log 103 contain information defining (1) the table being changed, (2) the value of the key column in the row being changed, (3) the old and new values of all columns of the changed row, and (4) the transaction (unit of work) containing the change. Recovery log entries for inserted rows contain only new column values while recovery log entries for deleted rows contain only old column values. Recovery log entries for updated rows contain the new and old values of all row columns. The order of entries in the recovery log reflect the order of change operations within each transaction and the order of transaction commit records reflects the order in which transactions are completed. The format of a row change log record can be abstracted as follows:

type	transid	tableId	old key cols	old non-key cols	new key cols	new non-key cols

To control the propagation of changes to table copies, copy control tables (not shown) designating table copies and their target table copies are used by the replication system. The control information includes, but is not limited to: (1) the name of the copied table, (2) a list of the table copies' key columns, (3) filtering and projection information, and (4) the message channels on which to send descriptions of changes to the target table copies.

The list of key columns defined for a replication definition will be hereafter referred to as the "replication key". The replication key should not be confused with other attributes of source or target table copies which may use primary key columns or foreign key columns.

However, it is possible that the primary key of a source or target table copy may be comprised of the same set of columns as are specified for the replication key. The replication key uniquely identifies a row entity in a target table copy so that it can be located by Apply, in applying an update or delete change operation. Because the replication key uniquely identifies a row entity, it is used in the serialization of changes made to these unique row entities.

The type of row operation in change log records can be delete, insert, update, or key update. Updates that do not modify the replication key (update) are distinguished from updates that do modify the replication key (key update).

The changes made to table copies are determined by reading the recovery log. Changes are saved in memory until a transaction commit record is seen on the recovery log. Only committed transactions at the source node 101 are moved and applied to target nodes 106. Change records are grouped into their originating source transaction units and written as one logical message unit. Because a logical transaction message can be quite large, it may be broken down into a plurality of physical messages. In this specification, a "transaction message" refers to a logical transaction message. Changes to be sent to the other table copies are sent via logical message units on the recoverable queues (e.g. send queue 105 and receive queue 107) designated in the copy control tables for the table copies of the log records.

The transactions messages are put on the recoverable queue in the source commit order. Within each transaction, the change records are arranged in the order in which they

occurred within the source transaction. In this embodiment, there is no inherent parallelism in the movement of the committed transactional data. The queuing of the transactional data is serialized such that data is moved to the target node 106 in the source transactional commit order.

In capturing the information for individual change records, the type of change operation for each change determines what replication key column values will be sent as part of that change record. For insert and update types of change records, the new replication key column values are sent as part of the change records within the transaction message. By definition, an insert is a new record and therefore has no old values. By definition, the new replication key column values of an update type of change record must be the same as the old replication key column values. For delete type change records, there is by definition no new record, only an old record, and therefore the old replication key column values are sent. For key update records, the old replication key column values are sent in addition to the new replication key column values.

Returning to Figure 1, for any given receive/recoverable queue 107 that is populated with transactions from a given source node 101 and is to be used as the source of changed data to be applied to a given target node 106, Apply 108 has a browser thread 109 and one or more agent threads 112, where the number of agents is determined through user input. The work queue 110 and the done queue 111, structures internal to Apply 108, are created for the purpose of communication between the browser thread 109 and the agent threads 112.

Figure 2 is a flowchart illustrating an embodiment of a method for providing parallel apply in asynchronous data replication in a database system in accordance with the present invention. First, the browser thread 109 examines the next transaction message in the receive queue 107, via step 201. The values of the replication key columns for each row change in the transaction message is remembered, via step 202. In this embodiment, information describing the transaction, including the values of the replication key columns, is remembered, i.e., stored in a logical data structure, and tracked. Other information concerning the transaction can also be remembered. The logical data structure also tracks any preceding non-completed transaction messages, including any subsequent transaction messages that are dependent upon it.

Next, the browser thread 109 determines if the transaction message has dependencies, via step 203. A transaction message has a dependency if the preservation of the integrity of the data requires that one or more preceding non-completed transaction messages be applied prior to the application of the current transaction message. If the transaction message has dependencies, then the browser thread 109 checks the transaction messages on the done queue 111 to see if the completion of any of those transaction messages clears the dependencies, via step 204. If not, then non-completed transaction messages upon which the transaction message is dependent are marked to indicate the transaction message's dependency, via step 205. The current transaction message is also marked with its dependencies and held, via step 206, and not allowed to be applied. If it does not have any dependencies, then the transaction message can be applied in parallel with

the preceding transaction(s) currently being applied, and is thus placed on the work queue 110, via step 207. Once placed on the work queue 110, the transaction message becomes eligible to be applied by any available agent thread 112. The more agent threads 112 that are made available to be used, the more transaction messages which are eligible for application can be applied in parallel.

Each of a plurality of agent threads 112 look on the work queue 110, and each removes a transaction message from the work queue, via step 208. Each agent thread 112 then applies the row changes in the transaction message to the target table copies 113 in parallel with each other, via step 209. All row changes from a transaction message are applied as a transaction unit, and are committed as a unit. In this embodiment, committed as part of this transaction is an update of a control table to indicate that this transaction has been successfully committed at the target table copy 113, via step 210. The update is an insert of an entry into the control table for the completed transaction. When the logical transaction message comprises a plurality of physical transaction messages, a plurality of entries, one for each physical transaction message, can be inserted. A control table in the same relational database as the target table copies 113 is used in order to provide for the best performance of this transaction application, while at the same time, keeping a permanent record of the successful application of the transaction. The insert to the control table is important for message cleanup of the receive queue 107, as described later in this specification.

In this embodiment, application of the changes is performed using generated Structured Query Language (SQL) statements of a non-proprietary nature. These SQL statements may or may not be exactly the same as the originating SQL statements made at the source node 101. However, the net effect of these changes is typically identical to the net effect of the changes made by the originating SQL statements. For example, an originating SQL statement such as "DELETE FROM SOURCE.TABLE" could be made. This statement would have the effect of deleting all rows from the table named SOURCE.TABLE. If there were five rows in the table at this point in time, then there would be five rows deleted, and five log records would be generated on the recovery log. Each log record would indicate the delete operation of one of the five rows. From the inspection of the recovery log, the five operations would be used to capture the information of five distinct data events, all of which occurred during a single transaction. This transaction would be queued and moved to the target node 106, and the application of these changes would be made as five distinct SQL statement, with SQL statement each targeting one of the individual rows of the corresponding target table copy. At the commit point of this applied transaction, the functional equivalence point is then reached, such that the same five rows have been deleted from the corresponding source and target table copies. Thus, the method and system in accordance with the present invention is a non-proprietary implementation of Apply. It could be extended for use in any database that accepts standard SQL and has the general database property of atomicity.

Once the application is complete, the transaction message is placed on the done queue 111, via step 211. The indicators of held transaction messages dependent on this now completed transaction message, if any exist, which were previously marked (via step 205) can now be checked, via step 212. These held transaction messages will be changed to remove the dependency or dependencies that existed regarding the now completed transaction message, via step 213. After removal of these dependencies, each of the held transaction messages are checked to see if any other dependencies remain, via step 214, against other preceding still non-completed transaction messages. Any held transaction message that is now determined to be dependency free, via step 214, can be safely applied in parallel with the other transaction messages currently being applied, and thus placed on the work queue 110, via step 207. For held transaction messages with remaining dependencies, they remain as held transaction messages.

Figure 3 is a flowchart illustrating in more detail the determining of dependencies in the method for providing parallel apply in asynchronous data replication in a database system in accordance with the present invention. For every transaction message that the browser thread 109 examines, critical pieces of information regarding that transaction are assessed and tracked. For each row change that makes up the transaction message, information regarding the values of the replication key columns is noted and tracked as part of that transaction. From the time of the initial examination of a transaction by the browser thread 109 until the eventual placement of that transaction message on the done queue 111 after successful application, the replication key column information for every row change

within this transaction message is used to assess newly arriving transactions, to determine their eligibility for placement on the work queue 110. If a newly assessed transaction message contains row changes with replication key column values that match the values of the replication key columns from row change of any preceding transaction messages that have not yet completed, then this newly assessed transaction message is not eligible yet for application and must not yet be placed on the work queue 110.

As illustrated in Figure 3, the browser thread 109 examines a transaction message in the receive queue, via step 301. The transaction message can contain a plurality of row changes. For each of the row changes, steps 302 through 312 are performed. The browser thread 109 examines the next change in the transaction message, via step 302. If the type of change is an insert or key update, via step 303, then the browser thread 109 determines if the new replication key value of the insert or key update change is the same as the old replication key value of any preceding non-completed transaction messages, via step 304. If they are the same, then the preceding non-completed transaction message is marked to indicate the transaction message's dependency, and the transaction message is marked to indicate the preceding non-completed transaction message upon which it depends , via step 305.

The new replication key column values of an insert or key update type of row change represent the introduction of a new row entity. Either of these row actions could have been preceded by a delete of that row entity (carrying old replication key column values) or by a key update which had the net effect of a delete followed by an insert, where it would be the

delete aspect of the prior row action that could potentially have commonality with this row action and is therefore of interest. Therefore, the new replication key column values of an insert or key update row change are compared to the old replication key column values of all preceding non-completed transaction messages.

The method by which it is determined that a new or old replication key value is the same as another new or old replication key value can be relaxed so long as the same replication key values are not determined to be different. Those with ordinary skill in the art at the time of the invention will recognize that the comparison of the result of any deterministic function (e.g., a hash code function) can be used to insure that identical replication key values are matched, while differing replication key values may be incorrectly matched. The performance benefits of simplified comparing can outweigh the loss of parallelism due to incorrectly matched replication key values.

If the type of change is a delete or a key update, via step 306, then the browser thread 109 determines if the old replication key value of the delete or key update change is the same as the new replication key value of any preceding non-completed transaction message, via step 307. If they are the same, then the preceding non-completed transaction message is marked to indicate the transaction message's dependency, and the transaction message is marked to indicate the preceding non-completed transaction message upon which it depends, via step 308.

The new replication key column values of an update type of row change represent the change of non-replication key column values of an existing row entity. This row action

could have been preceded by an insert of that row entity (carrying new replication key column values), or by a key update which had the net effect of a delete followed by an insert, where it would be the insert aspect of the prior row action that could potentially have commonality with this row action and is therefore of interest. Therefore, the new replication key column values of an update row change are compared to the new replication key column values of all preceding non-completed transaction messages.

If the type of change is an update, via step 309, then the browser thread 109 determines if the new replication key value of the update change is the same as the new replication key value of any preceding non-completed transaction message, via step 310. If they are the same, then the preceding non-completed transaction message is marked to indicate the transaction message's dependency, and the transaction message is marked to indicate the preceding non-completed transaction message upon which it depends, via step 311.

The old replication key column values of a delete or key update type of row change represent the deletion of an existing row entity. Either of these row actions could have been preceded by an insert of that row entity (carrying new replication key column values), by an update of that row entity (carrying new replication key column values), or by a key update which had the net effect of a delete followed by an insert, where it would be the insert aspect of the prior row action that could potentially have commonality with this row action and is therefore of interest. Therefore, the old replication key column values of a delete or key

update row change are compared to the new replication key column values of all preceding non-completed transaction messages.

Once the last change in a transaction message has been examined, via step 312, and the transaction message is determined to have dependencies, via step 313, the process continues with step 204 (Figure 2). If the transaction message is determined to have no dependencies, then the process continues with step 207 (Figure 2).

With the method in accordance with the present invention, whole source transactions are executed as whole target transactions, and changes to any individual table row entity, as determined by the specified and required replication key column values, are serialized to the same degree that those changes were serialized at the source database. Transactions with no dependencies are likely to be committed in a different order from the source commit order.

Figure 4 illustrates an example of the method for providing parallel apply in asynchronous data replication in a database system in accordance with the present invention. The transaction data found in the recovery log 103 is grouped by transaction and those transactions are sent to the send queue 105 in source commit order. For example, transaction 1 (Tx1), transaction 2 (Tx2), and transaction 3 (Tx3) were started in Tx1-Tx2-Tx3 order, but were committed in Tx1-Tx3-Tx2 order. Thus, they are sent to the receive queue 107 in committed Tx1-Tx3-Tx2 order.

When Tx1 arrives on the receive queue 107, the browser thread 109 examines Tx1, via step 201. Information concerning Tx1 is remembered, via step 202. Such information includes the fact that Tx1 involves an insert into table T1 of a row with replication key value

= 1. Since there are no preceding transactions, Tx1 has no dependencies, via step 203. Tx1 is thus placed on the work queue, via step 207.

As Tx1 is removed from the work queue, via step 208, and being applied, via step 209, the browser thread 109 examines Tx3, via step 201. Information concerning Tx3 is remembered, via step 202. Such information includes the fact that Tx3 involves a delete from table T1 of a row with replication key value = 1 and an insert into table T1 a row with replication key value = 2. The browser thread 109 determines that Tx3 has a dependency for table T1 delete, since the old replication key value of the delete (key=1) is the same as the new replication key value for the insert in Tx1, via step 307. Assuming that Tx1 has not yet completed, there are no transaction messages on the done queue 111 so steps 204 and 205 are not performed. Tx1 is thus marked to indicate the dependency of Tx3, and Tx3 is marked to indicate it is dependent upon Tx1, via step 308. Tx3 is held, via step 206.

The browser thread 109 next examines Tx2 after it arrives on the receive queue 107, via step 201. Information concerning Tx2 is remembered, via step 202. Such information includes the fact that Tx2 involves an update in table T2 of a row with replication key = 1, and an update in table T2 of a row with replication key = 3. The browser thread 109 determines that Tx2 has no dependencies, via step 203 (and step 310), and places Tx2 on the work queue 110, via step 207.

When application of Tx1 completes, via step 209, the control table is updated to indicate its completion, via step 210. Tx1 is also placed on the done queue 111, via step 211. From the marks added to Tx1 above, the browser thread 109 knows to remove from

Tx3 its dependency upon Tx1. The browser thread 109 then checks if Tx3 is now dependency free, via step 212. Since Tx3 is now dependency free, it is placed on the work queue, via step 207.

In this embodiment, the receive queue 107 is a persistent queue, while the work queue 110 and the done queue 111 are not. The persistence of the receive queue 107 is to protect the integrity of the data in case of a system failure or some other interruption during the transaction application process. However, the persistent nature of the receive queue 107 requires that messages in the receive queue 107 be removed after transactional messages have been successfully applied. Otherwise, if the process is interrupted, the system upon restart will attempt to apply the changes in the transaction messages on the receive queue 107 again, leading to errors.

One possible method of removal is a two-phase commit approach, where the delete of the message from the receive queue 107 is committed as part of the same transaction at the target node 106 that applies the changes. Another method is to use an asynchronous "cleanup" approach, as described below. The asynchronous cleanup approach has the advantage of defraying the delay and overhead costs associated with the two-phase commit approach.

In the asynchronous cleanup approach, it is noted that a control table is updated and committed as part of the transaction that applies the changes associated with a logical replication transaction message at a target node 106. This allows for a background task to be executed on a periodic basis which deletes messages from the receive queue 107 based on

the existence of an entry in the control table indicating that this message has been successfully applied. After the delete of one or more logical transaction messages from the receive queue 107 has been committed, entries for the logical transmission message from the control table can be safely removed. If the logical transaction message comprises a plurality of physical transaction message, then each physical transaction has its own entry in the control table. Each entry for the physical messages is individually removed. This approach avoids the cost of a two-phase commit since the control table rows are deleted after the committed delete of the messages on the receive queue 107. If entries in the control table exist without corresponding queue messages because those messages have already been deleted due to some process interruption, this poses no possible harm to the system, and such extra control table rows can be safely removed at anytime.

An improved method for providing parallel apply in asynchronous data replication in a database system has been disclosed. The improved method and system provides a high speed parallel apply of transactional changes to a target node such that the parallel nature of the application of changes does not compromise the integrity of the data. The method and system detects, tracks, and handles dependencies between transaction messages to be applied to the target node. If a transaction message has a dependency on one or more preceding transaction messages whose applications have not yet completed, that transaction message is held until the application completes. In addition, the method and system requires significantly less overhead than conventional approaches and is easily adaptable to various types of database systems.

Although the present invention has been described in accordance with the embodiments shown, one of ordinary skill in the art will readily recognize that there could be variations to the embodiments and those variations would be within the spirit and scope of the present invention. Accordingly, many modifications may be made by one of ordinary skill in the art without departing from the spirit and scope of the appended claims.